Living with Concurrency

Peter Grogono

CUSEC 2008

It's a pleasure and a privilege to be speaking to you today. I have given several previous talks at CUSEC and have always found the audiences to be demanding and responsive. For those of you who do not live in Montreal, let me assure you that the weather is not always like this. As well as the winter, we have a blackfly season and a mosquito season. Seriously... the summers are great!

Let's get on with the talk.

⇓ It's funny how, when you're asked to give a talk and can't think of what to say, some inspiring item always seems to pop up. For me, it was a recent article by Bob Dewar and Ed Schonberg, both professors emeriti of New York University. Dewar is president, and Schonberg is vice-president, of AdaCore.

Bob and Ed are worried about the decline of computer science education. Math requirements are shrinking; programming skills are declining; and the industry is worried. They quote

⇓ Bjarne Stroustrup, who has heard complaints from industry about students who were taught Java as their ***first programming language***. He switched EE students to C++ and the CS department followed. Bjarne is, of course, prejudiced: he's the chief architect of C++.

Whenever you hear professors emeriti, or any other old geezers, talking about education, you should be wary. If you believed every statement about teaching and students made by middle-aged people, you would have to conclude that education has been steadily declining since Socrates taught Plato. Especially beware of arguments like this:

⇓ I took this from a university website but changed the name of the university in case they sue me. You will notice that they offer the usual good things — analytical skills, communication skills, study habits — so that students end up with "real-world careers". (Are there careers in virtual worlds?) And all this is based on a study of assembler and C++!

I have to confess to altering the text there: the web site did not say "assembler and C++". Any guesses as to what it ***did*** say? Well, actually it said "Latin and Greek".

The picture is ***not*** the same university and is put there just to suggest a Gothic way of thinking.

This is the point. Do we "discipline minds" using any old set of ideas — Latin, Greek, assembler, whatever — or do we teach useful and applicable skills?

Let's get back to computer science. Is CS education declining. Certainly not! Let me illustrate from my own experience.

⇓ Here's the computer that I used to run my first program as a university student. It's EDSAC II. You can see that us guys all wore tweed jackets then. You punched your program on paper tape, fed the tape into the machine, as the woman at left is doing and, if you were lucky, the machine punched another tape containing your results. It was hard to get time on the machine because Fred Hoyle was always using it for astrophysics.

There were no CS programs, and I was doing — or "reading", as they called it — mathematics. We had one programming course, and the only program I can remember writing read a list of numbers and produced a sorted list. The lecturer was

⇓ Maurice Wilkes, seen here checking the mercury delay lines for EDSAC, the predecessor of EDSAC II and the first computer ever built with a true von Neumann architecture, meaning that both data ***and instructions*** were stored in memory. Those mercury delay lines are in fact the memory: information was stored in the form of ripples that flowed along the tank.

It's hard to believe that people really worked like that, but I suspect that they might be posing for the picture.

That photo was taken around 1948. Wilkes is still around, although

⇓ he looks a bit older now. It's impressive that the guy who designed the first computer is still around and still having good ideas. Amongst other things

⇓ he can claim to be the inventor of debugging. It's a pity that his lectures to undergraduates were rather boring.

Anyway, we've come a long way since then. I'm frequently impressed by the elaborate programs that students come up with — we've come a long way from sorting! Most schools now teach Java as the introductory programming language. I've taught Pascal, C++, and Java to first-year students, and my conclusions are shown here.

⇓ Java has several big things going for it. One is that it is simple enough that students can learn programming without being lost in a maze of semicolons, asterisks, ampersands, and segmentation faults. (Does anyone remember what they are?) Another is that you can't really do anything with Java without its massive collection of libraries, and so you pretty quickly get to learn how to use libraries — useful experience for professionals! Finally, students ***enjoy*** using Java, because they get immediate visual feedback.

But Java is a relatively "high level" programming language. Most assignments in early courses don't need non-trivial algorithms. It doesn't have pointers — well, actually it does, but only in the sense that a car with automatic transmission has a gearbox: it's there but you don't need to know about it. When students come to my C++ course, many admit to being "scared of pointers".

Another problem with high-level languages is that there is no relation between how hard it is to program something and how hard it is to compute it. Let's look at that — the issue of cost — in a little more detail.

⇓ Oscar Wilde said it first. In fact, Wilde said most things first — unless Shakespeare had already said them. Perlis cleverly reversed it. (Perlis was a very clever man.) He's saying that LISP is an expression language — so everything has a value — but it's also high-level — things that look simple can be expensive to compute.

⇓ I could not resist this illustration of Perlis's aphorism, because I think it's so neat. It's a dozen lines of Haskell that implement formal operations on infinite power series.

⇓ Here's an example of its use. We can define the function $y = e^x$ as the function that is its own derivative — $\frac{dy}{dx} = y$ — which has the value 1 at the origin. Integrating gives the second line. The third line shows the Haskell translation, using McIlroy's functions. Evaluating `expx` gives the coefficients of the power series for e^x.

⇓ It gets better. We can use the derivatives of `sin` and `cos` to write the mutually recursive definitions on the 3rd and 4th lines. Evaluation gives the power series for `sin` and `cos` respectively.

I was so impressed when I saw this code that I downloaded it and tried it out. McIlroy suggested generating 30 terms of the series. I tried that... and ran out of heap space. Easy enough to correct — the default heap was only a megabyte — but a bit of a surprise when a dozen line of code and "small" calculation end up using megabytes!

The morale: high-level languages are great fun but, the higher the level, the harder it is to estimate costs.

⇓ That's the same McIlroy, by the way, who proposed "software components" for "software engineering" back in 1968.

But let's get back to the story: what should you, as a student, be learning?

⇓ The language doesn't matter much: a professional programmer codes in the language chosen for, or most suited to, the project.

It helps to be able to work with objects, functions, data structures and the corresponding algorithms; have a basic idea about hardware. And, increasingly, know about concurrency.

⇓ I'm addressing only technical issues, of course. There are many other things that a software engineer has to know.

⇓ What's so important about concurrency?

⇓ Here's a graph. The horizontal axis is time — 1971 through 2007. The vertical axis is logarithmic. Little green squares are transistor counts, increasing exponentially for 30 years. Moore's law in action!

Little blue squares are clock speeds, increasing exponentially until 2003, then levelling off.

From 1970 until 2003 we programmers enjoyed what Herb Sutter calls "a free lunch". We could make software more and more complex, without worrying too much about

efficiency, because exponentially improving hardware would compensate for our sloppiness.

The hardware guys are still getting more transistors, but they can't speed the clock up. What have they done about that? Multicore!

⇓ To old fogies like me, "multicore" refers to the kind of solder that we used to assemble audio amplifiers in the days when people built their own amplifiers and called it "hi-fi". But that was then and now it's "wi-fi".

⇓ Today, multicore means "more CPUs on a chip". This is an IBM chip with 8 cores or, in IBM jargon, "synergistic processor elements". SIMD is a standard abbreviation for "single instruction, multiple data". This chip is specialized for graphics and games and, with single precision arithmetic, can achieve 200 GFlops.

⇓ The big problem is that concurrency is hard.

Mamihlapinatapai is a Yaghan word meaning "a look shared by two people with each wishing that the other will initiate something that both desire but which neither one wants to start".

⇓ Here's a popular way of avoiding concurrency problems. Use the three-tier architecture for your application, with canned middleware for the business logic. The top (presentation) layer is a conventional GUI, and the bottom layer is just a database.

⇓ All the concurrency is in the middle layer. Most of it is written by experts.

⇓ Dave Thomas calls this "Jurassic middleware". Maybe it's not such a neat solution after all.

⇓ Here's what Edward Lee has to say about concurrency. Lee is the Robert S. Pepper Distinguished Professor Chair of Electrical Engineering and Computer Sciences University of California at Berkeley.

How are we going to meet Lee's requirements for more deterministic systems?

⇓ This is my view of the software development process. It's a funnel. Why?

The top of the funnel is very broad because requirements come in all shapes and sizes. Every kind of software is included.

All applications are eventually expressed in machine code: that's why the bottom of the funnel is narrow. Just above the bottom is a layer called "PL" for programming languages. At any given time, a small number of programming languages are very popular. These languages determine the prevailing ***paradigm***.

⇓ The programming paradigm gradually takes over the whole funnel, from the bottom up. In the bad old days, we did not know better than to write spaghetti code and develop with the waterfall model.

(Aside: Royce's original paper on the waterfall model is much cited but rarely read. It's much better than the people who denigrate it realize. Few people used the waterfall model in the simpistic way in which it is usually understood anyway.)

Structured code did away with spaghetti by introducing nice control structures — if and while and so on — and encouraged SADT. After a brief flirtation with modules, OOPLs took over and triggered OOAD. Objects morphed into ASPECTS.

⇓ Here's the implication of the funnel analysis: to change the development process, start by changing the programming paradigm.

⇓ Here's my prediction: the next paradigm will be based on process-oriented languages and will lead to a new development methodology based on processes and models.

Let's call it "POMDD".

⇓ Our hypothesis is that POMDD, in some form or another, will succeed. First, because software must model the world, and the consists of concurrent processes. Secondly, because we can achieve the SE goals of lower coupling and refactoring with concurrent processes.

What are these "cells"? That comes next.

⇓ I will conclude this keynote with a few words about the project that I am working on — Erasmus. If you heard my CUSEC keynote a couple of years ago, this is your update.

⇓ I am working on Erasmus with my old friend, Brian Shearing, another EDSAC II user.

⇓ Erasmus programs consist of ***cells***, shown here as green boxes with round corners. The code at the bottom declares a ***cell type*** `Main` and creates an instance of it.

In this and the next few slides, the code can be compiled and executed. The ability to compile and run incomplete programs allows prototyping.

⇓ Cells may contain other cells — the green box on the right — and processes — the red rectangle with sharp corners on the left. Cells and process communicate using ports and channels. To fix the direction of communication, we label servers with + and clients with −. The code shows how the entities are linked together.

⇓ The type and sequence of messages are determined by ***protocols***. The protocol at the top specifies a `start` signal followed by any number of `query`/`reply` exchanges. A `stop` signal terminates communication and, in this case, the process. Protocols have the same structure as regular expressions, and we can use standard techniques to reason about them.

⇓ In the next development step, we put a process inside the client cell. And so it goes. Let's look at some other features of the language.

⇓ Here are some protocols. In `query`, the caret specifies a response, sent by a server to a client.

Protocols subsume two important aspects of OOP. First, we can model a ***method*** with a protocol that allows one or more arguments to be passed and a result to be returned. This generalizes easily to the case where there is more than one result.

Second, we can model a ***class*** with a protocol that accepts method invocations in any sequence.

This shows that protocols are at least as powerful as methods and classes in OOP but it does not imply that that is how we should use protocols or that it is all we can do.

⇓ We can choose which messages we want to process using a `select` statement. Most concurrent PLs have something similar.

The branches may have ***guards***, as shown in the middle. Guards are just Boolean expressions that must be `true` for the guard to be selected.

Finally, we can impose ***policies*** that determine the order in which the branches are checked.

⇓ Here's a process. The main point about this one is the statement `filter(q)` about half way down. This is a dynamic, and incidentally recursive, invocation of the process.

⇓ Recursive processes are a bit hard to diagram, but we can do better...

⇓ The + is an ***external*** port and the − is an ***internal*** port.

⇓ Here's a small but complete application with two processes, each nested inside a cell.

⇓ Here's the source code for this application. You can compile it and it runs on a single processor.

⇓ If you give the compiler this metacode as well as the source code, the compiled program runs on ***two*** processors. We do not have to alter the original program to do this: communication code is inserted by the compiler at the appropriate places.

⇓ Here are a few important points about cells. "Control flow never crosses a cell boundary" — that's important because it distinguishes Erasmus from OOPLs in which one object can directly call methods in another object.

⇓ The last line — processes are co-routines — is important, too. We will return to it in a moment.

⇓ Here are some important points about processes. Note that processes in the same cell can share variables.

⇓ The co-routine property of cells implies that there is only one program counter per cell, here shown as # in cell `C1`.

In `C1`, processes `P1` and `P2` share the variable `V1`. But there are no race conditions because `P1` and `P2` execute as co-routines, sharing a single processor, rather than concurrently.

`P3`, however, can run concurrently, because it is in another cell. But `P1` and `P2` cannot access its variable, `V2`.

⇓ This is about protocols. Again, the last point is important. When a server is connected to a client, we do not require that the protocols are the same. Instead, we require that the server protocol ***satisfies*** the client — the server can do all that the client needs. This can be checked statically, by the compiler.

⇓ Communications look just like assignments. More precisely, a sent message is an lvalue (left of assignment operator) and a received message is an rvalue (right of assignment operator).

⇓ Received messages really are rvalues and can be used in expressions.

⇓ The Erasmus model supports software engineering goals by ***separating concerns***.

Cells determine the architecture of a program; processes determine what it does.

Code defines the meaning of a program; metacode defines how it is executed — single processor, multiprocessor, network, ...

Protocols have a dual role: they act as interfaces to processes, and they determine communication patterns between processes.

⇓ When I describe this project, I usually get a number of objections. Here are the usual ones.

⇓ It's been done before. Quite true: we feel that it is time to resurrect some good ideas from the past and reuse them in the light of recent experience.

⇓ Our programming languages are fine; the problem is management, complexity, people, documentation, ...

This objection is wrong because PLs are the software engineer's basic tool. We do not yet have good languages. There's lots of room for improvement.

⇓ The OO paradigm is not broken; don't fix it.

This objection is wrong because the OO paradigm is already too complex is becoming more so — that's another talk.

When I've convinced people that OO is not good enough, they claim that ...

⇓ aspects will be good enough. Wrong again, same reason.

⇓ Present practice is fine because we can hide the hard bits (recall the three-tier architecture.

That's a short-term kludge, not a solution.

⇓ It's easy to make a case that a new paradigm means too much change. I have two counter-arguments:

20 years ago I listened to a bunch of UNIX wizards arguing that UNIX offers so much — operating system, compilers, libraries, applications, the kitchen sink — that it could never be replaced. Look what happened — Windows!

Will we be using OOP with threads in 2050? Probably not. When should we start thinking about the transition? Now!

⇓ I think there are many arguments in favour of a process-oriented programming paradigm.

⇓ First, there are many concurrent applications already and there will be many more.

⇓ Second, we are going to have to live with multicore processors whether we like it or not.

⇓ Third, I think POMDD is good — but that's a whole other talk and I haven't got time for the details now.

⇓ But here are some of the reasons.

⇓ That's it — thank you.